Growing Twigs

An Introduction to XML::Twig

Content

- What is XML::Twig
  - A description of the module, why use it
- Working with XML::Twig
  - Resources, Installation, Example code
- Behind the scenes
  - Development process, why open-source

What is XML::Twig

- XML::Twig: XML, The Perl Way
  - a Perl module
  - to process XML
  - hybrid processing model, perlish API
- Alternatives
  - Perl Modules: XML::LibXML, XML::Simple, XML::SAX
  - XSLT
  - Java, Python, Ruby...

A Perl Module

- a Perl Module is a library that can be used from a perl program
- most perl modules (several 1000s) can be found on CPAN (http://cpan.org)
- like a lot of modules, XML::Twig is Object Oriented:

```
use XML::Twig;
my $twig= XML::Twig->new( @arguments);
```

processing XML

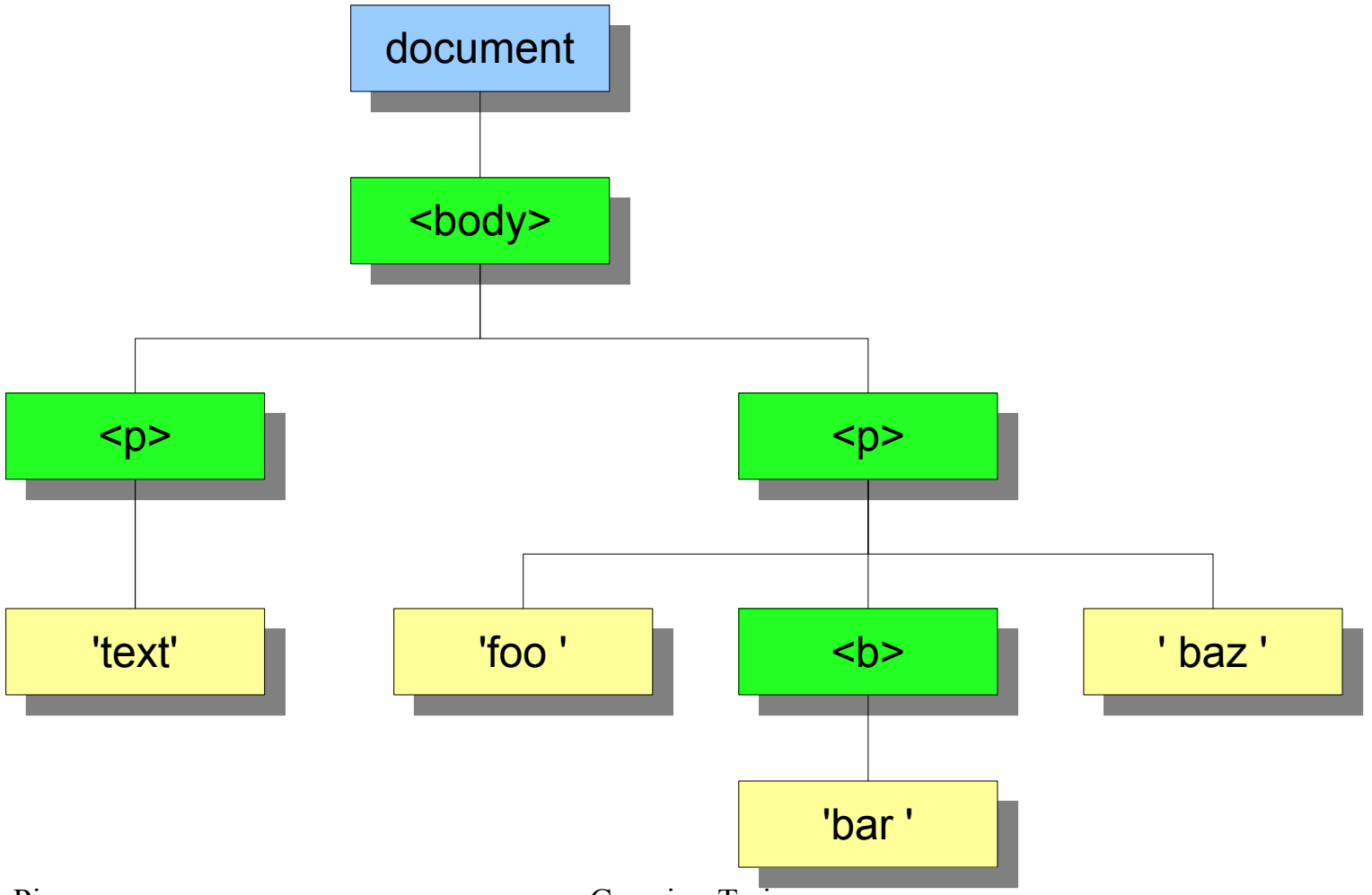
- XML::Twig can parse XML and process it
- I use it to:
  - generate XML from IEEE Standards in FrameMaker
  - generate XHTML from IEEE Standards in XML
  - extract definitions from IEEE Standards in XML and store them into a data base
  - move data between databases on different OSs
  - power the templating system for my wife's website
  - ...

XML Processing Models

- Stream Mode (SAX)
  - during parsing, call methods for each parsing event (open tag, text, close tag)
  - low memory usage, complex to use
- Tree Mode (DOM)
  - load the XML in memory, as a tree of objects
  - select nodes using navigation or queries (Xpath)
  - transform using delete, move, insert methods
  - needs more memory, easier to use

XML Tree Model

```
<body><p>text</p><p>foo <b>bar</b> baz</p></body>
```



XML::Twig Processing Model

- Tree mode, using a simplified DOM
- Possibility to add handlers to elements
  - selected by element name, or complex condition,
  - called when the element is finished parsing
  - handler has access to the tree for the element
- Possibility to build the tree only for certain elements
  - other elements are ignored or output as-is

Other XML::Twig features

XML::Twig is designed to be practical

- whitespace handling
- comment/processing instructions handling
- encoding handling
- rich API (over 500 methods)

Other Perl Modules

- XML::LibXML
  - based on `libxml2` (http://xmlsoft.org)
  - very powerful, fast, supports Xpath, DOM and lots of other W3C standards
- XML::Simple
  - converts data-oriented XML to a Perl data structure
- XML::SAX
  - event-driven, low-level
  - lots of helper modules

XSLT

- W3C's language for processing XML
- Works well
- The code is in XML
- You don't get CPAN!

Working with XML::Twig

- Installing XML::Twig
  - installing the pre-requisites: `expat`, `XML::Parser`
- Resources
  - finding information on how to use the module
- Example
  - a (semi!) realistic example of code written using XML::Twig: updating data from a big XML file.

Installing XML::Twig

- Pre-requisites:
  - perl! (5.005 minimum, 5.8.3+ recommended)
  - `expat`: the low-level XML parsing library
  - `XML::Parser`: the Perl wrapper for `expat`
  - optional Perl modules (`XML::XPath`, `LWP`, `HTML::Entities`)

Installing Perl Modules

- the old-fashioned way

```
tar zxvf XML-Twig-3.22.tar.gz
perl Makefile.PL
make
make test
make install
```

- cpan / cpanplus

```
cpan XML::Twig
```

- distribution packages

```
urpmi perl-XML-Twig
```

Resources

- The `README` file

  install instructions, dependencies, links

- `perldoc XML::Twig`

  reference doc

- `http://xmltwig.com`

  docs, tutorial, FAQ, examples, development version

- `http://perlmonks.org`

The Most Important Slide

- Always, ALWAYS, check the data first:
  - parse the XML before doing anything with it
  - if you can, refuse the XML if it is not valid
  - if you cannot, write code to fix it, then validate it
- It doesn't matter who generated the XML, another company, another department, your department, YOU ...
- ONLY work on clean data
- You WILL hate character encodings

Examples presentation

Data-oriented vs Document-oriented XML

- Text is messy, data is simpler!
- Data has more structure
- Data has no mixed-content
- Differences in usage
- Some tools work best (or only!) for data-oriented XML
- Most XML these days is data-oriented

Data-oriented XML

- Data Base dumps/extracts
- Standard Documents
- Serialized objects
- XML-RPC
- log files
- Configuration files

Time

for a

quick

break!

Stop

Using

<XML>

Everywhere

Please!

XML is Everywhere

- Documents
- Configuration files
- Data
- Serialized objects

Configuration Files

- XML is turned into a Perl Data Structure
- Works reasonably

but...

XML is UGLY!

XML Version

```
<config logdir="/var/log/foo/"
        debugfile="/tmp/foo.debug">
  <server name="sahara" osname="solaris"
          osversion="2.6">
    <address>10.0.0.101</address>
    <address>10.0.1.101</address>
  </server>
  <server name="gobi" osname="irix"
          osversion="6.5">
    <address>10.0.0.102</address>
    <address>10.0.0.103</address>
  </server>
</config>
```

YAML Version

```
Debugfile: '/tmp/foo.debug'
logdir: '/var/log/foo/'
server:
  gobi:
    address:
      - 10.0.0.102
      - 10.0.0.103
    osname: irix
    osversion: 6.5
  sahara:
    address:
      - 10.0.0.101
      - 10.0.1.101
    osname: solaris
    osversion: 2.6
```

Try this at home

```
perl -MYAML -MXML::Simple \
-e 'print Dump XMLin "conf.xml"'
```

XML for Data

Data lives in...

Data Bases!

Data Bases

- Fast
- Reliable
- Multi-user
- Scalable!

XML for data

It's just like text files...

...only **slower!**

Exporting XML

- XML::Generator::DBI
- XML::Handler::YAWriter

```
use DBI;
use XML::Generator::DBI;
use XML::Handler::YAWriter;

my $dbh= DBI->connect(...);

my $ya = XML::Handler::YAWriter->new(AsFile => "-");
my $generator = XML::Generator::DBI->new(
                    Handler => $ya, dbh => $dbh
                                        );
$generator->execute('SELECT * FROM data');
```

Conclusion

- Use XML when it makes sense
- Don't use it just because it's a buzzword

THINK!

Typical use of data-oriented XML

XML is an EXCHANGE format

- Extract data
  - Put it in a Data Base
- Fix the data
- Add data
- Avoid XML transformations!
  - if it's data, it should live in a DATA BASE

Example 1: XML Catalog

```
<?xml version="1.0" encoding="utf-8"?>
   <catalog>
     <plant id="id_001">
       <common>Bloodroot</common>
       <botanical>Sanguinaria canadensis</botanical>
       <zone>4</zone>
       <light>Mostly Shady</light>
       <price>$2.44</price>
       <availability>2005-03-05</availability>
     </plant>
    ...
     <plant id="id_036">
       <common>Cardinal Flower</common>
       <botanical>Lobelia cardinalis</botanical>
       <zone>2</zone>
       <light>Shade</light>
       <price>$3.02</price>
       <availability>2005-02-05</availability>
     </plant>
   </catalog>
```

Example 1

- Store records from the catalog in a table in a database

Example 1: code

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
use XML::Twig;

my $CATALOG_FILE = "plant_catalog.xml";
my $DB_FILE      = "plant_catalog.db";

my $dbh= DBI->connect("dbi:SQLite:dbname=$DB_FILE","","");
my $sth= $dbh->prepare( "INSERT into plant
              ( id, common, botanical, zone, light, price, availability)
         VALUES( ?,  ?,      ?,         ?,    ?,     ?,     ?)");

XML::Twig->new( twig_handlers => { plant => \&store_plant })
         ->parsefile( $CATALOG_FILE);

sub store_plant
  { my( $t, $plant)= @_;
    $sth->execute( $plant->id, map { $_->text } $plant->children);
    $t->purge;
  }
```

Example 2: convert currency

Convert the prices in dollars to prices in euros, add the currency as an attribute:

```
<price>$3.02</price>
```

becomes

```
<price currency="EUR"> 2.58</price>
```

The code will be a filter that will only update the `price` elements and leave everything else untouched.

Example 2: code

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::Twig;

my $CATALOG_FILE = "plant_catalog.xml";

# a silly example of extracting information from a web page
my $RATE = XML::Twig->nparse( "http://www.x-rates.com/index.html")
                    ->first_elt( 'a[@href="/d/USD/EUR/graph120.html"]')
                    ->text;
warn "rate: 1 EUR = $RATE USD\n";

my $catalog=  XML::Twig->new( twig_roots => { price    => \&price, },
                              twig_print_outside_roots => 1,
                            );

$catalog->parsefile( $CATALOG_FILE);

exit;
```

Example 2: code *(cont.)*

```
sub price

  { my( $twig, $price)= @_;

    my $value= $price->text;

    if( $value=~ /^\$(.*)$/)
      { my $dollar_value= $1;
        my $euro_value= sprintf( "%5.2f", $dollar_value / $RATE);
        $price->set_text( $euro_value);
        $price->set_att( currency => "EUR");
      }
    else
      { die "wrong dollar value '$value'\n"; }

    $price->print;

  }
```

Example 3: Update the data

- Update the catalog file with data from an other file
- 2 input XML files:
  - catalog
  - updates
- Output: updated catalog file
- The update file can be loaded in memory, not the main catalog

Example 3: XML update

```
<updates>
   <plant id="id_013">
     <price>$7.22</price>
   </plant>
   <plant id="id_033">
     <availability>2005-05-28</availability>
   </plant>
   <plant id="id_021">
     <price>$4.20</price>
     <availability>2005-05-08</availability>
   </plant>
 </updates>
```

Example 3: code

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::Twig;

my $CATALOG_FILE = "plant_catalog.xml";
my $UPDATE_FILE  = "updates.xml";

my $updates= XML::Twig->new->parsefile( $UPDATE_FILE);

my $catalog=  XML::Twig->new(    # element => subroutine
              twig_handlers => { plant    => \&plant, },
              pretty_print  => 'indented',
                             );

$catalog->parsefile( $CATALOG_FILE);
$catalog->flush;

exit;
```

Example 3: code *(cont.)*

```
sub plant
  { my( $twig, $plant)= @_;

    my $id= $plant->att( 'id');
    my $update= $updates->elt_id( $id);  # updates is global

    if( $update)
      { foreach my $updated ( $update->children)
          { my $field    = $updated->tag;
            my $original = $plant->first_child( $field);
            $original->replace_with( $updated);

            warn "updating $id - $field: ", $original->text,
                                       " => ", $updated->text, "\n";
          }
      }
    $twig->flush; # prints the XML so far, and frees the memory
  }
```

Document-oriented XML

- Important in publishing
- Allows:
  - independence from vendors
  - re-purposing of documents or parts of documents
- Often include embedded data
- Often used to generate HTML or PDF

Processing document-oriented XML

- Need to be able to work at 4 levels:
  - document level: to grab cross-references, number clause titles... often in a separate pass,
  - complex element level: tables, lists with internal references, chapter,
  - simple element processing: change a tag into an other tag (often adding the initial element as a class attribute),
  - within text: generate links from URLs, or from text elements, parse element text or attribute values.

Document Example

```
<?xml version="1.0" encoding="utf-8"?>
  <plant id="id_001">
    <common>Bloodroot</common>
    <botanical>Sanguinaria canadensis</botanical>
    <zone>4</zone>
    <light>Mostly Shady</light>
    <price>$2.44</price>
    <available>2005-03-05</available>
    <desc>A perennial <i>native</i> with a solitary white
        flower with golden stamens around a solitary pistil on a smooth
        stalk. 5-10 inches tall, this early plant has a reddish-orange
        juice down to the root (hence the name). The large blue/grey
        to green basal leaf is palmately scalloped into 5-9 lobes. See
        http://www.main.nc.us/naturenotebook/plants/bloodroot.html and
        http://en.wikipedia.org/wiki/Bloodroot
    </desc>
  </plant>
```

HTML generation code

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::Twig;
use Regexp::Common 'URI';

my $PLANT_FILE="plant.xml";

my $twig= XML::Twig->new(
  twig_handlers => {
    common    => sub { $_->set_tag( 'h1') },            # $_ is the element
    botanical => sub { $_->set_tag_class( 'p') },       # set tag to 'p' and
    zone      => sub { $_->set_tag_class( 'p');         # class to the tag
                       $_->prefix( "Grows in zone ");
                     },
    light     => sub { $_->set_tag_class( 'p');
                       $_->prefix( "Required Light: ");
                     },
    price     => sub { $_->set_tag_class( 'p'); },
    available => sub { $_->set_tag_class( 'span');
                       $_->prefix( ", available ");
                       $_->move( last_child => $_->prev_sibling);
                     },
```

HTML generation code *(cont.)*

```
 desc       => sub { $_->set_tag_class( 'p');
                     $_->insert_new_elt( before => h2 => "Description");
                     $_->subs_text( qr/($RE{URI}{HTTP})/,
                                    '&elt( a =>{ href => $1 }, $1)'
                                  );
                   },
   plant      => sub { $_->set_tag( 'body'); },
                 },
  pretty_print => 'indented',
                    );

$twig->parsefile( $PLANT_FILE);

# add the html "wrapping"
my $html= $twig->root->wrap_in( 'html');
my $head= $html->insert_new_elt( first_child => 'head');

my $name= $twig->first_elt( 'h1')->text;
$head->insert_new_elt( first_child => 'title', $name);

$twig->print;
```

Behind the scenes

The history of XML::Twig

- Why did I write XML::Twig?
- Why is it Open-Source?
- Development Process
- ToDo list

Why did I write XML::Twig

- Timeline:
  - 1998-02-10: the XML recommendation is published
  - 1998-03-??: XML::Parser published on CPAN
  - 1998-10-??: XML::Twig development starts
  - 1998-10-21: XML::DOM on CPAN
  - 1999-10-04: XML::Twig 1.6 on CPAN
  - 2005-10-14: XML::Twig 3.22 on CPAN
- In 1998 there were no XML module that would do what I wanted, I had to write my own!

Why is it Open-Source?

- Instead of having to find the bugs, people (sometimes!) find them for me
- A good way to give back to the Open Source community that gave me Linux, Apache, PostgreSQL, SQLite, vi, Firefox, OpenOffice... and Perl!
- It's fun!

Development Process

- It has evolved with time:
  - in 1998 there was no Test Driven Development
- Now:
  - revision control (CVS)
  - tests added for every bug and new feature (Devel::Cover used to check coverage)
- Still very much a Cathedral, not a Bazaar

ToDo List

- Write a proper Xpath parser

  needs to be used both in streaming mode, to trigger handlers and in normal mode, on an element or document

- Add “multi-parsing”

  start several parsers (in threads) and allow them to rendez-vous to perform actions on all of them

  example: merging sorted XML files

The End

Questions?

Grazie